Chapter 22

Working with Arrays

This chapter introduces fundamental algorithms for arrays—sorting and searching—and then finishes with a sample application that uses arrays.

22.1 Sorting Arrays

Array sorting—arranging the elements within an array into a particular order—is a common activity. For example, an array of integers may be arranged in ascending order (that is, from smallest to largest). An array of words (strings) may be arranged in lexicographical (commonly called alphabetic) order. Many sorting algorithms exist, and some perform much better than others. We will consider one sorting algorithm that is relatively easy to implement.

The selection sort algorithm is relatively simple and efficient. It works as follows:

1. Let A be the array and let $i = 0$.
2. Examine each element that follows position i in the array. If any of these elements is less than $A[i]$, then exchange $A[i]$ with the smallest of these elements. This ensures that all elements after position i are greater than or equal to $A[i]$.
3. If $i < A.\texttt{length} - 1$, then set i equal to $i + 1$ and goto Step 2.

If the condition in Step 3 is not met, the algorithm terminates with a sorted array. The command to "goto Step 2" in Step 3 represents a loop. The directive at Step 2 to "Examine each element that follows ..." must also be implemented as a loop. Thus, a pair of nested loops is used in the selection sort algorithm. The outer loop moves a pointer from position zero to the next-to-the-last position in the array. During each iteration of this outer loop, the inner loop examines each element that follows the current position of the outer loop's pointer.

`SortIntegers` (22.1) sorts the list of integers entered by the user on the command line.

```
public class SortIntegers {
    public static void selectionSort(int[] a) {
        for ( int i = 0;  i < a.length - 1;  i++ ) {
            int small = i;
            //  See if a smaller value can be found later in the array
```



```
            for ( int j = i + 1;  j < a.length;  j++ ) {
                if ( a[j] < a[small] ) {
                    small = j;  //  Found a smaller value
                }
            }
            //  Swap a[i] and a[small], if a smaller value was found
            if ( i != small ) {
                int tmp = a[i];
                a[i] = a[small];
                a[small] = tmp;
            }
        }
    }
    public static void print(int[] a) {
        if ( a != null ) {
            for ( int i = 0;  i < a.length - 1;  i++ ) {
                System.out.print(a[i] + ", ");
            }
            //  No comma following last element
            System.out.println(a[a.length - 1]);
        }
    }
    public static void main(String[] args) {
        //  Convert the arrays of strings into an array of integers
        int[] numberList = new int[args.length];
        for ( int i = 0;  i < args.length;  i++ ) {
            numberList[i] = Integer.parseInt(args[i]);
        }
        selectionSort(numberList);   //  Sort the array
        print(numberList);           //  Print the results
    }
}
```

Listing 22.1: `SortIntegers`—Sorts the list of integers entered by the user on the command line

The command line

```
java SortIntegers 4 10 -2 8 11 4 22 9 -5 11 45 0 18 60 3
```

yields the output

```
-5, -2, 0, 3, 4, 4, 8, 9, 10, 11, 11, 18, 22, 45, 60
```

`SortIntegers` uses its `selectionSort()` method to physically rearrange the elements in the array. Since this `selectionSort()` method is a public class method (it is `static` and therefore cannot access any instance variables), it can be used by methods in other classes that need to sort an array of integers into ascending order. Selection sort

is a relatively efficient simple sort, but more advanced sorts are, on average, much faster than selection sort. One such general purpose sort is *Quicksort*, devised by C. A. R. Hoare in 1962. Quicksort is the fastest sort for most applications. Since sorting is a common activity, Java provides a standard library class, `java.util.Arrays`, that implements Quicksort. Among other useful methods, the `Arrays` class contains a number of overloaded `sort()` class methods that can sort many different types of arrays. To use this standard Quicksort in the `SortIntegers` program (22.1), simply replace the statement

```
selectionSort(numberList);  //  Sort the array
```

with

```
java.util.Arrays.sort(numberList);  //  Sort the array
```

or, if `java.util.Arrays` is imported, just

```
Arrays.sort(numberList);  //  Sort the array
```

The results will be the same, but for large arrays, the Quicksort version will be much faster. `CompareSorts` (22.2) compares the execution times of our selection sort to Quicksort.

```
import java.util.Random;
import java.util.Arrays;

public class CompareSorts {
    public static void selectionSort(int[] a) {
        for ( int i = 0;  i < a.length - 1;  i++ ) {
            int small = i;
            //  See if a smaller value can be found later in the array
            for ( int j = i + 1;  j < a.length;  j++ ) {
                if ( a[j] < a[small] ) {
                    small = j;  //  Found a smaller value
                }
            }
            //  Swap a[i] and a[small], if a smaller value was found
            if ( i != small ) {
                int tmp = a[i];
                a[i] = a[small];
                a[small] = tmp;
            }
        }
    }
    public static void main(String[] args) {
        //  Program must have size to be able to run
        if ( args.length != 1 ) {
            System.out.println("Usage:");
            System.out.println("        java CompareSorts <array_size>");
            System.exit(1);
        }
        //  Allocate the arrays from size provided on the command line
        int[] listSS = new int[Integer.parseInt(args[0])];
```



```
        int[] listQS = new int[listSS.length];
        //  Create a random number generator
        Random random = new Random();
        //  Initialize the arrays with identical random integers
        for ( int i = 0;  i < listSS.length;  i++ ) {
            int randomValue = random.nextInt();
            listSS[i] = listQS[i] =randomValue;
        }
        long timeSS = 0, timeQS = 0;
        Stopwatch timer = new Stopwatch();
        //  Time selection sort
        timer.start();
        selectionSort(listSS);
        timer.stop();
        timeSS = timer.elapsed();
        timer.start();
        //  Time Java's Quicksort
        Arrays.sort(listQS);
        timer.stop();
        timeQS = timer.elapsed();
        //  Report results
        System.out.println("Array size = " + args[0]
                           + "   Selection sort: " + timeSS
                           + "   Quicksort: " + timeQS);

    }
}
```

Listing 22.2: `CompareSorts`—Compares the execution times of selection sort versus Quicksort

Table 22.1 shows the results of running `CompareSorts`.

Our selection sort is faster than Quicksort for arrays of size 500 or less. Quicksort performs much better for larger arrays. A 100,000 element array requires about four minutes to sort with our selection sort, but Quicksort can sort the same array in about one-fifth of a second! Section 22.2 addresses performance issues in more detail.

22.2 Searching Arrays

Searching an array for a particular element is a common activity. `LinearSearch` (22.3) uses a `locate()` method that returns the position of the first occurrence of a given element in a 1D array of integers; if the element is not present, -1 is returned.

```
public class LinearSearch {
    //  Return the position of the given element; -1 if not there
```


Array Size	Time in msec	
	Selection Sort	Quicksort
10	< 1	4
50	1	4
100	3	5
500	15	16
1,000	29	18
5,000	464	31
10,000	1,839	41
50,000	48,642	120
100,000	238,220	206

Table 22.1: Empirical data comparing selection sort to Quicksort. This data was obtained running `CompareSorts` (22.2). The results are averaged over five runs.

```
    public static int locate(int[] a, int seek) {
        for ( int i = 0;  i < a.length;  i++ ) {
            if ( a[i] == seek ) {
                return i;   //  Return position immediately
            }
        }
        return -1;  //  Element not found
    }
    //  Print an integer right-justified in a 4-space field
    private static void format(int i) {
        if ( i > 9999 ) {
            System.out.print("****");
            return;
        }
        if ( i < 1000 ) {
            System.out.print(" ");
            if ( i < 100 ) {
                System.out.print(" ");
                if ( i < 10 ) {
                    System.out.print(" ");
                }
            }
        }
        System.out.print(i);
    }
    //  Print the contents of the array
    public static void print(int[] a) {
        for ( int i : a ) {
            format(i);
        }
        System.out.println();  //  newline
    }
    //  Tab over the given number of spaces
    private static void tab(int spaces) {
```



```
        for ( int i = 0;  i < spaces;  i++ ) {
            System.out.print("    ");
        }
    }
    private static void display(int[] a, int value) {
        int position = locate(a, value);
        if ( position >= 0 ) {
            print(a);
            tab(position);
            System.out.println("   ^");
            tab(position);
            System.out.println("   |");
            tab(position);
            System.out.println("   +-- " + value);
        } else {
            System.out.print(value + " not in ");
            print(a);
        }
        System.out.println();
    }
    public static void main(String[] args) {
        int[] list = { 100, 44, 2, 80, 5, 13, 11, 2, 110 };
        display(list, 13);
        display(list, 2);
        display(list, 7);
        display(list, 100);
        display(list, 110);
    }
}
```

Listing 22.3: `LinearSearch`—finds an element within an array

Running `LinearSearch` produces

```
 100  44   2  80   5  13  11   2 110
                     ^
                     |
                     +-- 13

 100  44   2  80   5  13  11   2 110
           ^
           |
           +-- 2

7 not in  100  44   2  80   5  13  11   2 110

 100  44   2  80   5  13  11   2 110
   ^
```



```
   |
   +-- 100

100  44   2  80   5  13  11   2 110
                                 ^
                                 |
                                 +-- 110
```

The key method in the `LinearSearch` class is `locate()`; all the other methods simply lead to a more interesting display of `locate()`'s results. Examining each method reveals:

- `locate()`— The `locate()` method begins at the beginning of the array and compares each element with the item sought.

```
//  Return the position of the given element; -1 if not there
public static int locate(int[] a, int seek) {
    for ( int i = 0;  i < a.length;  i++ ) {
        if ( a[i] == seek ) {
            return i;   //  Return position immediately
        }
    }
    return -1;  //  Element not found
}
```

  If a match is found, the position of the matching element is immediately returned; otherwise, if all the elements of the array are considered and no match is found, -1 is returned (-1 can never be a legal index in a Java array).

- `format()`—`format()` prints an integer right-justified within a four-space field. Extra spaces pad the beginning of the number if necessary.

- `print()`—`print()` prints out the elements in any integer array using the `format()` method to properly format the values. This alignment simplifies the `display()` method.

- `tab()`—`tab()` prints four blank spaces and leaves the cursor on the same line. It is used by `display()` to position its output.

- `display()`—`display()` uses `locate()`, `print()`, and `tab()` to provide a graphical display of the operation of the `locate()` method.

The kind of search performed by `locate()` is known as *linear* search since a straight line path is taken from the beginning to the end of the array considering each element in order. Figure 22.1 illustrates linear search.

Linear search is acceptable for relatively small arrays, but the process of examining each element in a large array is time consuming. An alternative to linear search is *binary search*. In order to perform binary search an array must be in sorted order. Binary search takes advantage of this organization by using a clever but simple strategy that quickly zeros in on the element to find:

1. Let

   - *A* be the array,

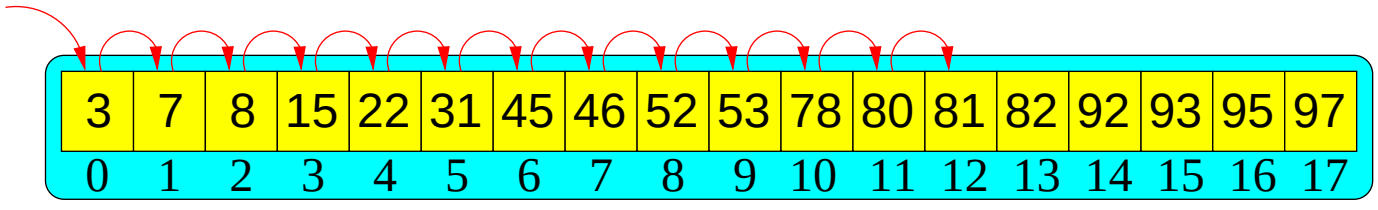

Figure 22.1: Linear search of an array. The algorithm makes 13 probes before value 81 is found to be at index 12.

- n be the length of A, and
- x be the element to find.

2. Find the middle position, m, of A. $m = \lfloor \frac{n}{2} \rfloor$.
3. If $A[m] = x$, the element is found and the answer is m; stop.
4. If $n = 1$, the element is not present and the answer is -1; stop.
5. Otherwise,
   - if $A[m] > x$, then continue the search using the same strategy with a new array consisting of elements $0 \ldots m-1$ of the current array, or
   - if $A[m] < x$, then continue the search using the same strategy with a new array consisting of elements $m+1 \ldots n-1$ of the current array.

This approach is analogous to looking for a telephone number in the phone book in this manner:

1. Open the book at its center. It the name of the person is on one of the two visible pages, look at the phone number.
2. If not, and the person's last name is alphabetically less the names of the visible pages, apply the search to the left half of the open book; otherwise, apply the search to the right half of the open book.
3. Discontinue the search with failure if the person's name should be on one of the two visible pages but is not present.

This algorithm can be converted into a Java method:

```
//  Return the position of the given element; -1 if not there
public static int binarySearch(int[] a, int seek) {
    int first = 0,            //  Initially the first element in array
        last = a.length - 1,  //  Initially the last element in array
        mid;                  //  The middle of the array
    while ( first <= last ) {
        mid = first + (last - first + 1)/2;
        if ( a[mid] == seek ) {
            return mid;       //  Found it
        }
        if ( a[mid] > seek ) {
```



```
            last = mid - 1;   //  last decreases
        } else {
            first = mid + 1;  //  first increases
        }
    }
    return -1;
}
```

In `binarySearch()`:

- The initializations of `first` and `last`:

  ```
  int first = 0,        //  Initially the first element in array
  last = a.length - 1,  //  Initially the last element in array
  ```

  ensure that `first` $\leq$ `last` for a nonempty array. If the array is empty, then

  $$(\texttt{first} = 0) > (\texttt{a.length} - 1 = -1)$$

  and the loop body will be skipped. In this case -1 will be returned. This is correct behavior because an empty array cannot possibly contain any item we seek.

- The calculation of `mid` ensures that `first` $\leq$ `mid` $\leq$ `last`.

- If `mid` is the location of the sought element (checked in the first `if` statement), the loop terminates.

- The second `if` statement ensures that either `last` decreases or `first` increases each time through the loop. Thus, if the loop does not terminate for other reasons, eventually `first` $>$ `last`, and the loop will terminate.

- The second `if` statement also excludes the irrelevent elements from further search.

Figure 22.2 illustrates how binary search works.

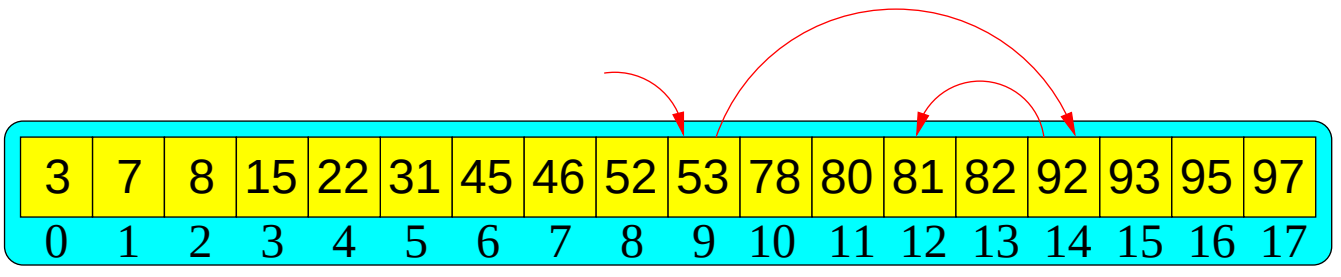

Figure 22.2: Binary search of an array. Compare this algorithm's three probes to locate the value 81 to the 13 probes required by linear search in Figure 22.1.

The Java implementation of the binary search algorithm is more complicated than the simpler linear search algorithm. Ordinarily simpler is better, but for algorithms that process data structures that can potentially hold large amounts of data, more complex algorithms that employ clever tricks that exploit the structure of the data (as binary search does) often dramatically outperform simpler, easier-to-code algorithms. `SearchCompare` (22.4) compares the performance of linear search versus binary search on arrays of various sizes.


```
import java.util.Random;
import java.util.Arrays;

public class SearchCompare {
    //  Return the position of the given element; -1 if not there.
    //  Note modification for ordered array---need not always loop to
    //  very end.
    public static int linearSearch(int[] a, int seek) {
        for ( int i = 0;  i < a.length && a[i] <= seek;  i++ ) {
            if ( a[i] == seek ) {
                return i;   //  Return position immediately
            }
        }
        return -1;  //  Element not found
    }
    //  Return the position of the given element; -1 if not there
    public static int binarySearch(int[] a, int seek) {
        int first = 0,            //  Initially the first element in array
            last = a.length - 1,  //  Initially the last element in array
            mid;                  //  The middle of the array
        while ( first <= last ) {
            mid = first + (last - first + 1)/2;
            if ( a[mid] == seek ) {
                return mid;       //  Found it
            }
            if ( a[mid] > seek ) {
                last = mid - 1;   //  last decreases
            } else {
                first = mid + 1;  //  first increases
            }
        }
        return -1;
    }

    //  Print the contents of the array
    public static void print(int[] a) {
        for ( int i : a ) {
            System.out.print(i + " ");
        }
        System.out.println();  //  newline
    }
    public static void main(String[] args) {
        //  Program must have size to be able to run
        if ( args.length != 2 ) {
            System.out.println("Usage:");
            System.out.println("        java SearchCompare <array_size>"
                               + " <iterations>");
            System.exit(1);
        }
        //  Allocate the arrays size provided on the command line
```



```
        int[] list = new int[Integer.parseInt(args[0])];
        //  Create a random number generator
        Random random = new Random();
        //  Initalize the array with random integers
        for ( int i = 0;  i < list.length;  i++ ) {
            list[i] = random.nextInt();
        }
        //  Sort the array
        Arrays.sort(list);
//          print(list);
        int linearTime = 0, binaryTime = 0;
        Stopwatch timer = new Stopwatch();
        //  Profile the searches under identical situations
        for ( int i = 0;  i < Integer.parseInt(args[1]);  i++ ) {
            int seek = random.nextInt();
            timer.start();
            linearSearch(list, seek);
            timer.stop();
            linearTime += timer.elapsed();
            timer.start();
            binarySearch(list, seek);
            timer.stop();
            binaryTime += timer.elapsed();
        }
        //  Report results
        System.out.println(args[1] + " searches on array of size "
                           + args[0] + ":");
        System.out.println("Linear search time: " + linearTime
                           + "  Binary search time: " + binaryTime);

    }
}
```

Listing 22.4: `SearchCompare`—Compares the performance of linear search versus binary search

The `SearchCompare` program requires two command line arguments:

1. The size of the array
2. The number of searches to perform.

An array of the given size is initialized with random values and then arranged in ascending order. A random search value is generated, and both linear and binary searches are performed with this value and timed. The accumulated times for the given number of searches is displayed at the end. Observe that the same search value is used over the same array by both the linear search and binary search methods to keep the comparison fair.

Note that the linear search method (`linearSearch()`) of `SearchCompare` has been optimized for ordered arrays to discontinue the search when a value greater than the value sought is encountered. Thus, given an array of size n, on average only $\frac{n}{2}$ comparisons are required to determine that a sought value is *not* present in the array. The original

version always requires *n* comparisons if the item sought is not present in the array. This optimization for linear search is only possible if the array is ordered.

Table 22.2 lists empirical data derived from running `SearchCompare`. Figure 22.3 plots the results for arrays with up to 1,000 elements.

Array Size	Time (msec) to perform 15,000 random searches	
	Linear Search	Binary Search
10	80	109
20	122	114
50	186	126
100	322	152
200	578	176
500	1394	185
1000	2676	197
5000	13251	227
10000	26579	252
50000	132725	283

Table 22.2: Empirical data comparing linear search to binary search. This data was obtained running `SearchCompare` (22.4). The number of searches is fixed at 15,000 per run. The results are averaged over five runs. Figure 22.3 plots this data up to array size 1,000.

The empirical results show that for very small arrays (fewer than 20 elements) linear search outperforms binary search. For larger arrays binary search outperforms linear search handily, and the gap between their performances widens dramatically as even larger arrays are considered. Consider an array with 100,000 elements. Binary search would require approximately six seconds to return a result; searching for the same value in the same array using linear search would typically take 36 minutes and 34 seconds!

To better understand these radically different behaviors, we can analyze the methods structurally, counting the operations that must be performed to execute the methods. The search methods are relatively easy to analyze since they do all the work themselves and do not call any other methods. (If other methods were called, we would need to look into those methods and analyze their structure as well.) Operations that we must consider consist of:

- Assignment—copying a value from one variable to another
- Comparison—comparing two values for equality, less than, greater than, etc.
- Arithmetic—adding, subtracting, multiplying, dividing, computing modulus, etc. of two values
- Increment and decrement—incrementing or decrementing a variable
- Array access—accessing an element in an array using the subscript operator

All of these operations each require only a small amount of time to perform. However, when any of these operations are placed within a loop that executes thousands of iterations, the accumulated time to perform the operations can be considerable.

Another key concept in our analysis is *search space*. Search space is the set of elements that must be considered as the search proceeds. In any correctly devised search algorithm the search space decreases as the algorithm progresses until either the element is found or it is determined that the element is not present in the remaining search space.

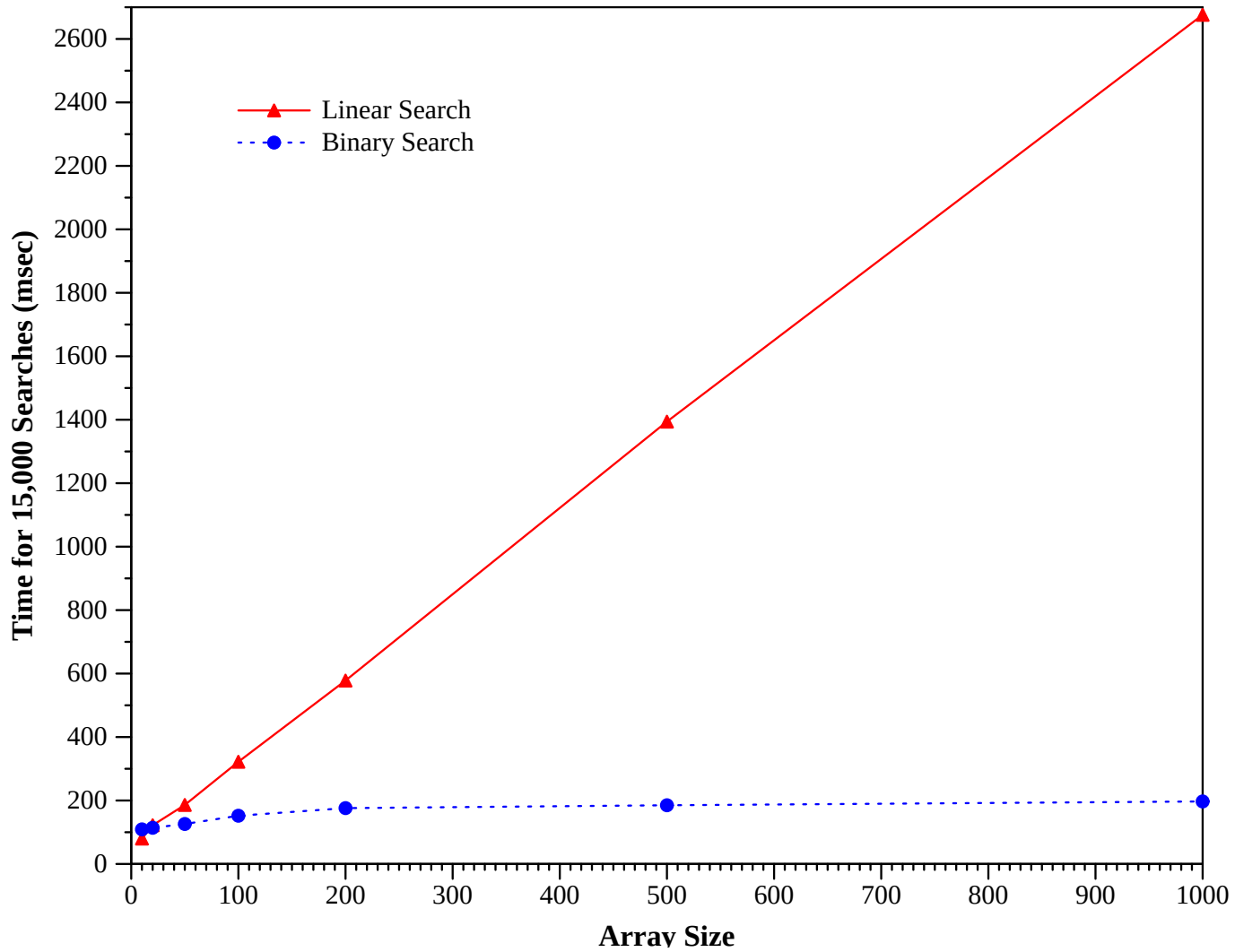

Figure 22.3: Comparison of linear search versus binary search. The results from Table 22.2 for array sizes up to 1,000 are plotted here. The vertical axis represents the number of milliseconds to make 15,000 random searches (averaged over five runs). The horizontal axis represents the number of elements in the array.

The key difference in the performance of linear search versus binary search is the manner in which the search space decreases as each algorithm executes.

Our analysis will be approximate since we are considering the Java source code, but the JVM is actually executing compiled bytecode, not the source code. A single source code expression may result in several bytecode instructions. Also, different types of instructions execute at different rates. For example, usually addition and subtraction operations can be performed faster than multiplication and division operations. For simplicity, we assume all operations require equal time to execute.

Table 22.3 tabulates the number of operations required when executing the `linearSearch()` method.

If the `for` loop is executed n times, then a total of $7n+2$ operations will be performed by the `linearSearch()` method. The number of loops performed depends on the size of the array being searched. On the average $n = \frac{N}{2}$ for an array of size N. This is because, given a random search value, the following is true:

- If the element is in the array, the loop begins at position zero, and considers each element in turn until the element is located. Each time through the loop the search space decreases by one element. The element is just as likely to be near the front as it is to be near the back. $\frac{N}{2}$ is the "average position" of an element in the array,

Linear Search	Analysis	Effort

`for ( int i = 0;`
`  i < a.length && a[i] <= seek;`
`  i++ ) {`
	`for` *loop header*	

`i = 0`	Initialization: performed once	1
`i < a.length`	Comparison: performed each time through loop	n
`a[i]`	Array access: performed each time through loop	n
`a[i] <= seek`	Comparison: performed each time through loop	n
`i < a.length && a[i] <= seek`	Logical *and*: performed each time through loop	n
`i++`	Incrementation: performed each time through loop	n
`if ( a[i] == seek ) {`	`if` *statement*	
`a[i]`	Array access: performed each time through loop	n
`a[i] == seek`	Comparison: performed each time through loop	n
`return i;`	Return: performed at most once	≤ 1
`}`	*End of* `if`	
`}`	*End of* `for`	
`return -1;`	Return: performed at most once	≤ 1
Total Operations	Exactly one of the `return` statements will be executed per method call, so both `return` statements count only as one	$7n+2$

Table 22.3: A structural analysis of the linear search method. The `for` loop is executed n times.

so $\frac{N}{2}$ iterations must be performed on average to find an element in an ordered array.

- If the element is not in the array, the loop begins at position zero, and considers each element in turn until the array element under consideration is greater than the item sought. Each time though the loop the search space decreases by one element. The "average position" of a missing element is also $\frac{N}{2}$ since the missing element is just as likely to belong in the first part of the array as the last part, so $\frac{N}{2}$ iterations must be performed on average to determine that the element is not present.

Since `linearSearch()` on average performs $\frac{N}{2}$ iterations given an array of size N, it must perform approximately $\frac{7}{2}N+2$ operations. This is a good measure of the amount of work `linearSearch()` must perform, and thus how long it will take to execute.

Now consider an analysis of the `binarySearch()` method as shown in Table 22.4.

If the `while` loop is executed m times, then a total of $12m + 4$ operations will be performed by the `binarySearch()` method. If n (the number iterations in `linearSearch()`) = m (the number of iterations in `binarySearch()`), then `linearSearch()` will clearly outperform `binarySearch()` for all array sizes. However, $n \neq m$ in general because a binary search traverses an array in a very different manner than linear search. As in the case of linear search, the number of loops performed by `binarySearch()` depends on the size of the array being searched. On the average, for an array of size N, $m = \log_2 N$ because, given a random search value:

Binary Search	Analysis	Effort
`first = 0`	Assignment: performed once	1
`last = a.length - 1`	*Assignment statement*	
`a.length - 1`	Subtraction: performed once	1
`last = a.length - 1`	Assignment: performed once	1
`while ( first <= last ) {`	`while` *loop header*	
`first <= last`	Comparison: performed each time through loop	m
`mid = first + (last - first + 1)/2;`	*Complex assignment statement*	
`last - first`	Subtraction: performed each time through loop	m
`(last - first + 1)`	Addition: performed each time through loop	m
`(last - first + 1)/2`	Division: performed each time through loop	m
`first + (last - first + 1)/2`	Addition: performed each time through loop	m
`mid = first + (last - first + 1)/2`	Assignment: performed each time through loop	m
`if ( a[mid] == seek ) {`	`if` *statement header*	
`a[mid]`	Array access: performed each time through loop	m
`a[mid] == seek`	Comparison: performed each time through loop	m
`return mid`	Return: performed at most once	≤ 1
`if ( a[mid] > seek ) {`	`if` *statement header*	
`a[mid]`	Array access: performed each time through loop	m
`a[mid] > seek`	Comparison: performed each time through loop	m
Exactly one of		
`last = mid - 1`	Subtraction and assignment (two operations)	
or		
`first = mid + 1`	Addition and assignment (two operations)	
is performed each time through loop		$2m$
`return -1`	Return: performed at most once	≤ 1
Total Operations	Exactly one of the `return` statements will be executed per method call, so both `return` statements count only as one	$12m+4$

Table 22.4: A structural analysis of the binary search method. The `while` loop is executed m times.

- If the element is in the array, the loop begins at the middle position in the array. If the middle element is the item sought, the search is over. If the sought item is not found, search continues over either the first half or the second half of the remaining array. Thus, each time through the loop the search space decreases by *one half*. At most $\log_2 N$ iterations are performed until the element is found.

- If the element is not in the array, the algorithm probes the middle of a series of arrays that decrease in size by one half each iteration until no more elements remain to be checked. Again, each time through the loop the search space decreases by *one half*. At most $\log_2 N$ iterations are performed until the remaining array is reduced to zero elements without finding the item sought.

Why $\log_2 N$? In this case $\log_2 x$ represents the number of times x can be divided in half (integer division) to obtain a value of one. As examples:

- $\log_2 1024 = 10$: 1,024 $\xrightarrow{1}$ 512 $\xrightarrow{2}$ 256 $\xrightarrow{3}$ 128 $\xrightarrow{4}$ 64 $\xrightarrow{5}$ 32 $\xrightarrow{6}$ 16 $\xrightarrow{7}$ 8 $\xrightarrow{8}$ 4 $\xrightarrow{9}$ 2 $\xrightarrow{10}$ 1

- $\log_2 400 = 8.6$: 400 $\xrightarrow{1}$ 200 $\xrightarrow{2}$ 100 $\xrightarrow{3}$ 50 $\xrightarrow{4}$ 25 $\xrightarrow{5}$ 12 $\xrightarrow{6}$ 6 $\xrightarrow{7}$ 3 $\xrightarrow{8}$ 1

For further justification, consider `Log2` (22.5) that counts how many times a number can be halved.

```
public class Log2 {
    private static int log_2(int n) {
        int count = 0;
        while ( n > 1 ) {
            n /= 2;
            count++;
        }
        return count;
    }
    public static void main(String[] args) {
        int badCalc = 0,
            goodCalc = 0;
        final int MAX = 100000;
        //  Compare halving to log base 2
        for ( int i = 1;  i < MAX;  i++ ) {
            if ( log_2(i) == (int)(Math.log(i)/Math.log(2)) ) {
                goodCalc++;
            } else {
                badCalc++;
            }
        }
        System.out.println("Agreements: " + goodCalc + "/" + (MAX - 1));
        System.out.println("Discrepencies: " + badCalc + "/" + (MAX - 1));
    }
}
```

Listing 22.5: `Log2`—demonstrates that $\log_2 x$ corresponds to determining how many times x can be divided in half

In `Log2`, the method `log_2()` determines how many times an integer can be divided by 2 until it is reduced to 1. This result is compared to finding the base 2 logarithm of that value. Java's `Math` class does not have a $\log_2$ method, but a basic mathematics identity shows us that given the logarithm of a particular base we can compute the logarithm of any other positive base:

$$\log_a x = \frac{\log_b x}{\log_b a}$$

Thus Java's `Math.log()` method (which is actually ln, see Section 13.5) is used to compute $\log_2$:

$$\frac{\log_e x}{\log_e 2} = \frac{\texttt{Math.log(}x\texttt{)}}{\texttt{Math.log(2)}} = \log_2 x$$

The results show total agreement:

```
Agreements: 99999/99999
Discrepencies: 0/99999
```

Since `binarySearch()` on average performs $\log_2 N$ iterations given an array of size N, it must perform approximately $12 \log_2 N + 4$ operations. As in the case of `linearSearch()` above, this is a good measure of the amount of work `binarySearch()` must perform, and thus how long it will take to execute.

Finally, we can compare our structural analysis to the empirical results. Figure 22.4 plots the mathematical functions over the various array sizes.

Despite our approximations that skew the curves a bit, the shape of the curves in Figure 22.4 agree closely with the empirical results graphed in Figure 22.3. The curves represent the *time complexity* of the algorithms; that is, how an increase in the size of the data set increases the algorithm's execution time. Linear search has a complexity proportional to N, whereas binary search's complexity is proportional to $\log_2 N$. Both graphs show that as the data size grows, the amount of time to perform a linear search increases steeply; linear search is impractical for larger arrays. For an application that must search arrays that are guaranteed to remain small (less than 20 elements), linear search is the better choice. Binary search, however, performs acceptably for smaller arrays and performs exceptionally well for much larger arrays. We say it *scales* well. Observe that both algorithms work correctly; sometimes successful applications require more than correct algorithms.

22.3 Summary

- Add summary items here.

22.4 Exercises

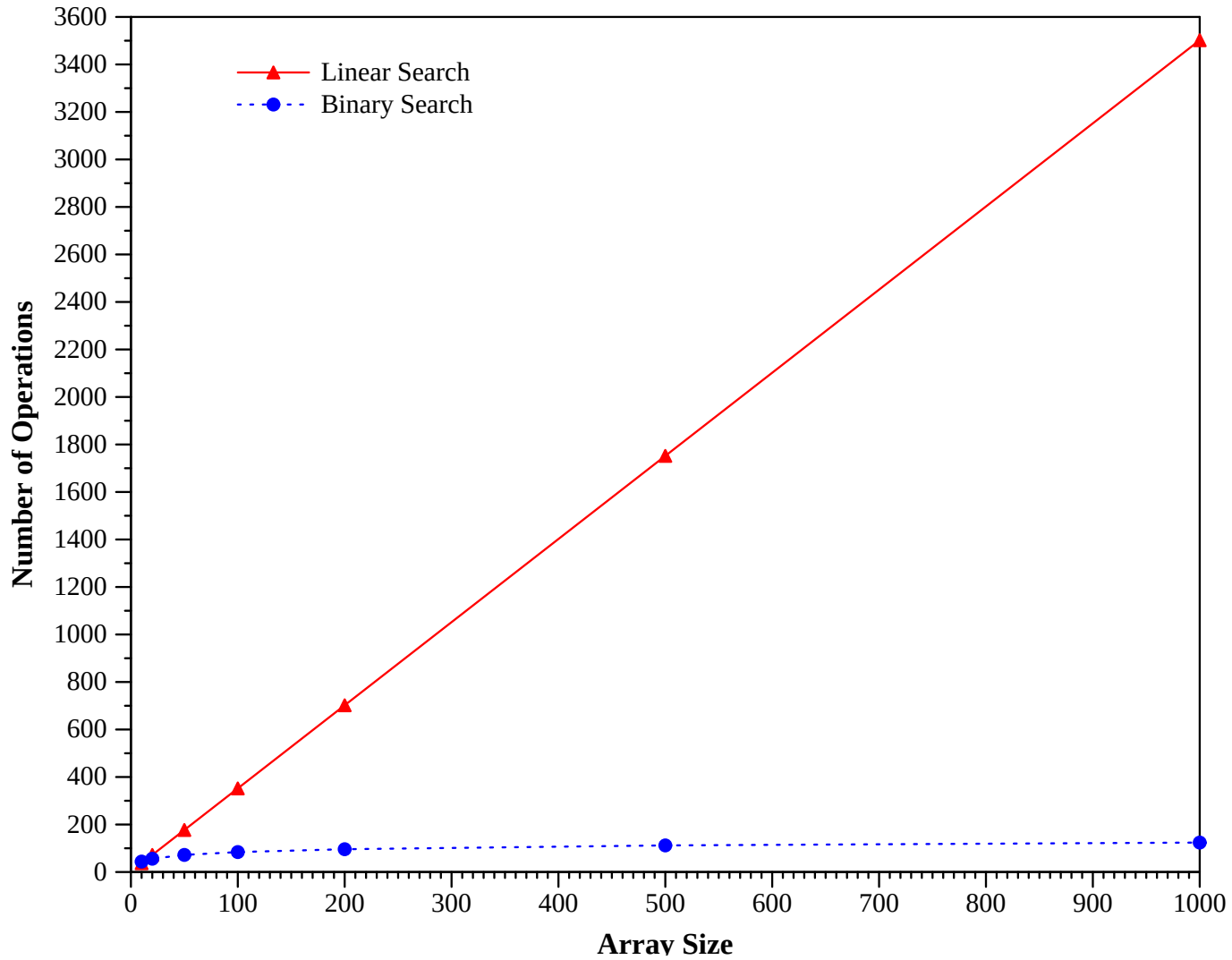

Figure 22.4: Plot of the mathematical functions obtained through the code analysis. The vertical axis represents the number of operations performed. The horizontal axis maps N, the size of the array. The linear search function is $\frac{7}{2}N+2$, and the binary search function is $12\log_2 N+4$.

